mustVisit - Project Final Report

Luca Ferrari - S4784573

Lorenzo La Corte - S4784539

Database Free Application

This app showcases an innovative concept that leverages the power of **large language models (LLM)** to eliminate the reliance on traditional databases.

Querying an LLM as GPT3.5 and parsing its outputs, the developer can avoid handling data gathering and maintenance.

Of course, this idea is not appliable to every situation, since it has different drawbacks:

PROs	CONs
Easy to implement	Slow to gather results
Extended knowledge of popular topics	Limited knowledge of niches
Cheap: no need to build and maintain a database	Infeasible if the query is too complex

To ensure responsible and effective application, certain scenarios are deemed ineligible for this concept.

For example, safety-critical applications, which demand absolute precision and minimal response times, should not adopt this approach; similarly, applications that revolve around a specific topic or require real-time communication with the backend might find this idea unsuitable.

However, **within the scope of our application, this mechanism proves to be a perfect fit**.

It presents an ideal use case to test and implement this innovative approach, offering users the benefits of simplified development, extended knowledge coverage, and cost-efficient data handling.

mustVisit Core Idea

> 💡 The idea is to present the top places to visit nearby the user location.

In particular:

1. The application locates the user with his current position: `(x, y)`.
2. The user specifies the range of kilometres: `range`.

   Example: (44.4076379, 8.9314594), with a range of 20 kilometres.
3. The user can select from the following checklist to be returned with a top list:
   - Historical places
   - Fun attractions
   - Beaches
   - Parks
4. **We use the chatGPT API to return a top list for each category**.
5. For each place, compute the distance between it and the current position:
   - If it is within the selected `range`, show it.
   - Otherwise, discard it.
6. Display all the places as lists with the following information:
   - `{Name of the place}`
   - `{Distance}`
   - `{Short Description}`
7. And also **visualize a map of the different places using Google Maps API**.
8. The user can then select some of the returned places and the system creates an itinerary for the user, redirecting him to **Google Maps**.

Showcase

This is a brief showcase of the main features of the application:

---

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/5c189950-2062-4ed4-b6c2-decd320d1e9d/Showcase.mp4

Use Case

> **Use Case**: Explore Top Attractions in a City

Description:
As a user of the application, I want to discover the top attractions in a city based on my current location and preferences.

Main Actors:

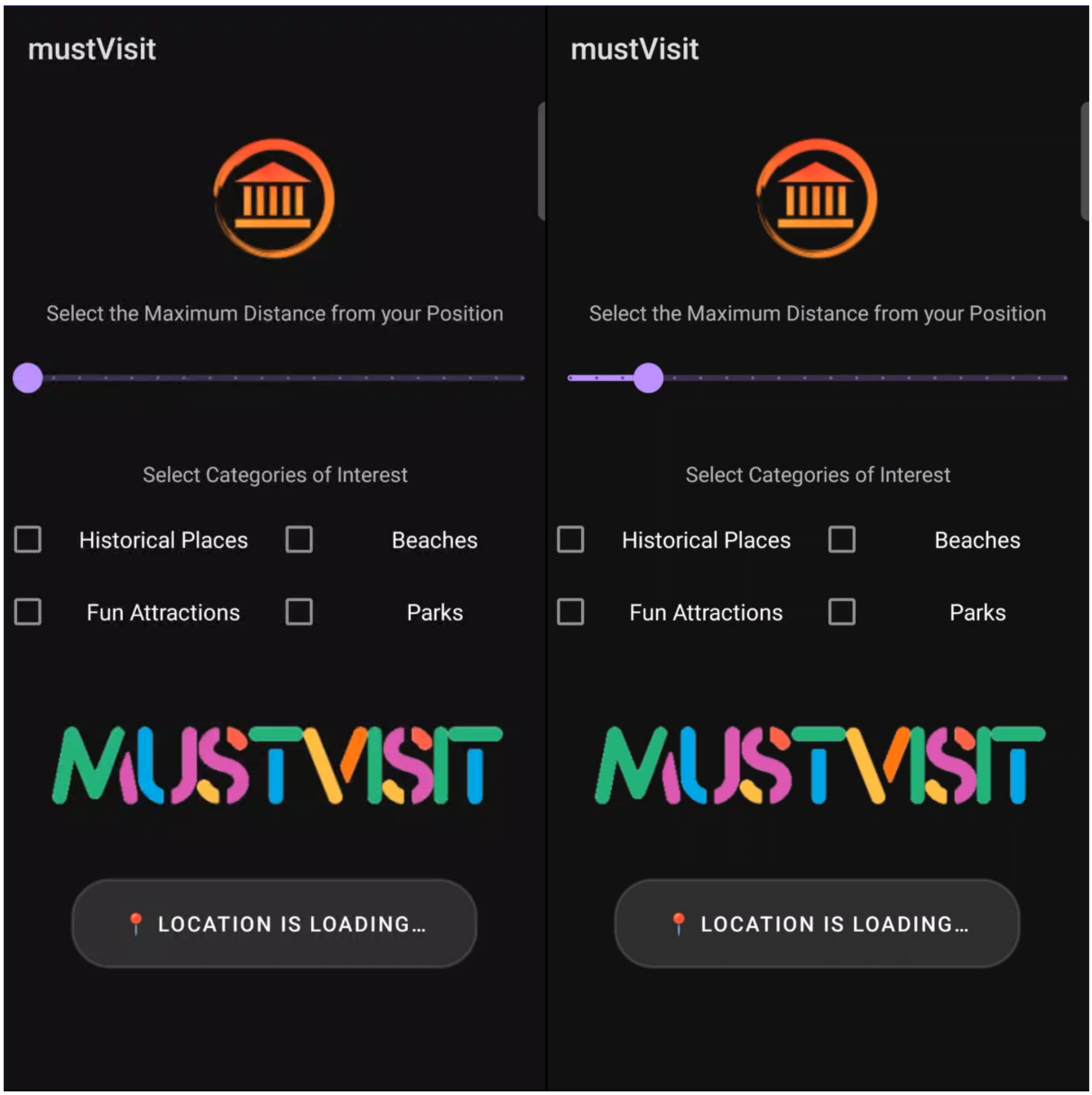
The person using the application to find top attractions.

Preconditions:

The user has been granted location access to the application.

Basic Flow:

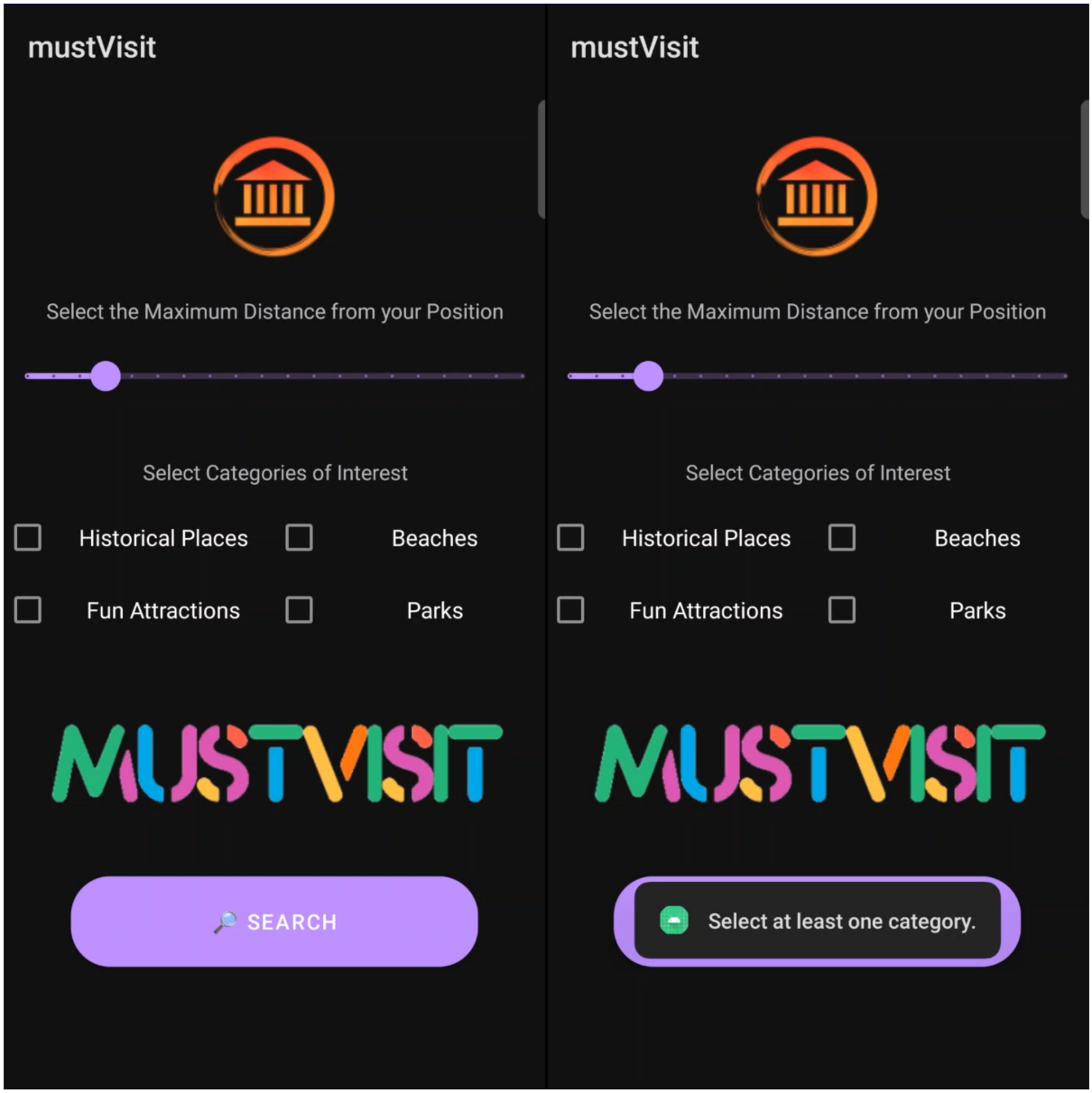
1. The application presents the user with its interface.
2. The user specifies their desired range in kilometres.



3. The user selects one or more categories from the checklist (e.g., historical places, fun attractions, beaches, parks, etc.).

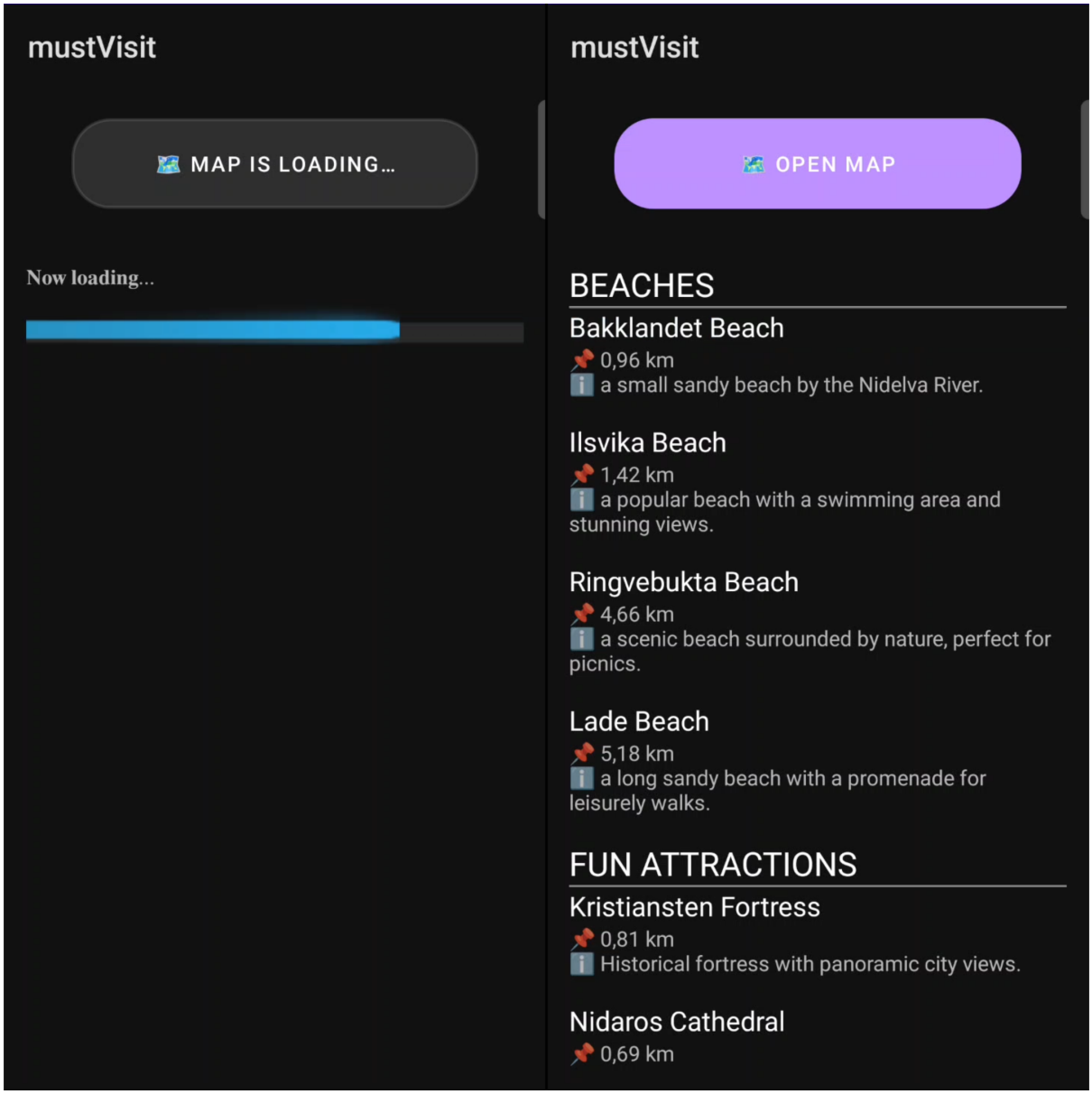
   Alternative Flow: If the user doesn't select any categories, an error is shown.

4. The user launches the application and locates himself, providing his current location (latitude and longitude).



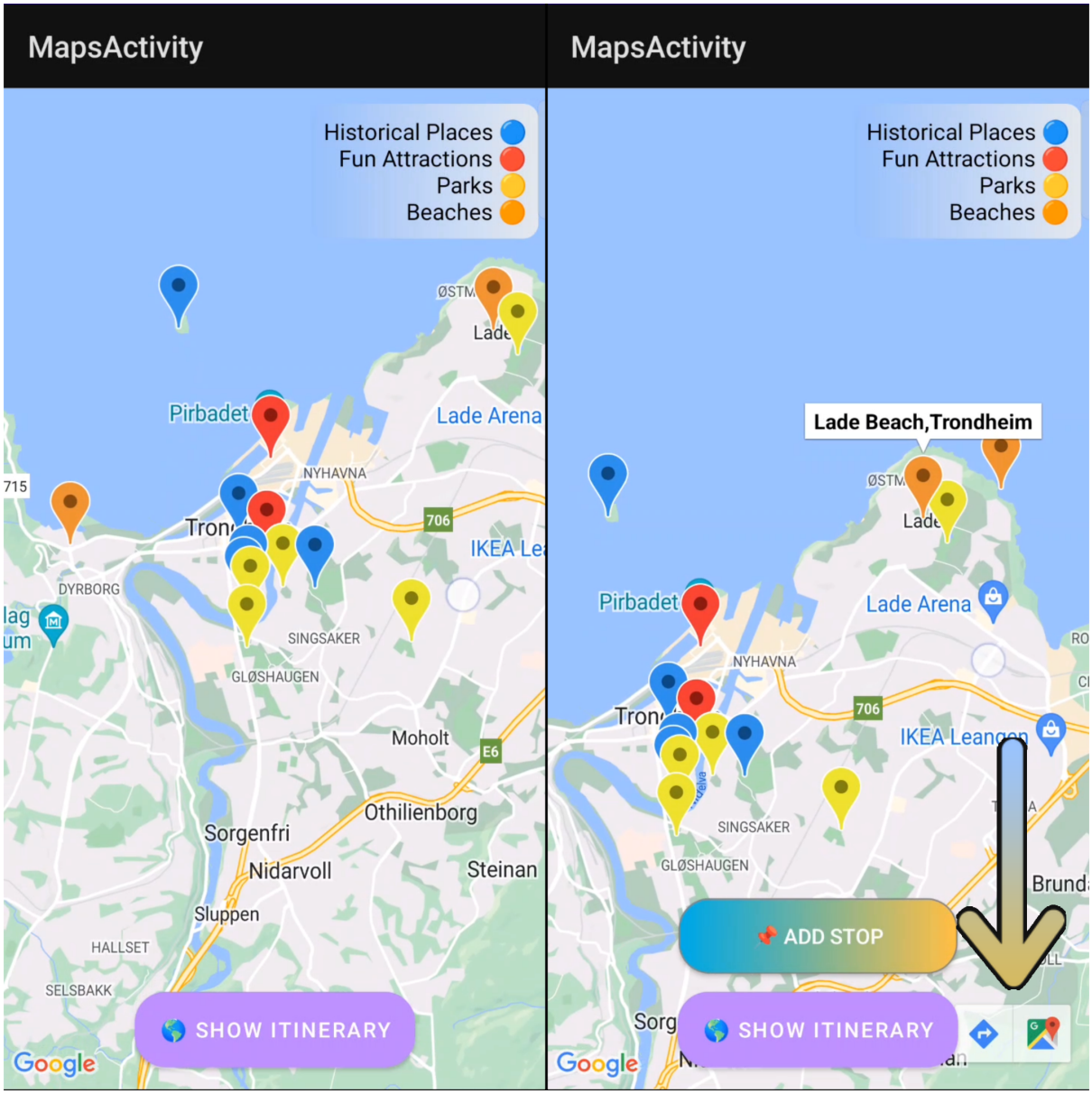
5. The application presents different boxes where the user can find lists of places, including the name, distance, and description for each place.

   Alternative Flow: If there are no results within the range, the system will retry the query 2 times before visualizing an empty box.

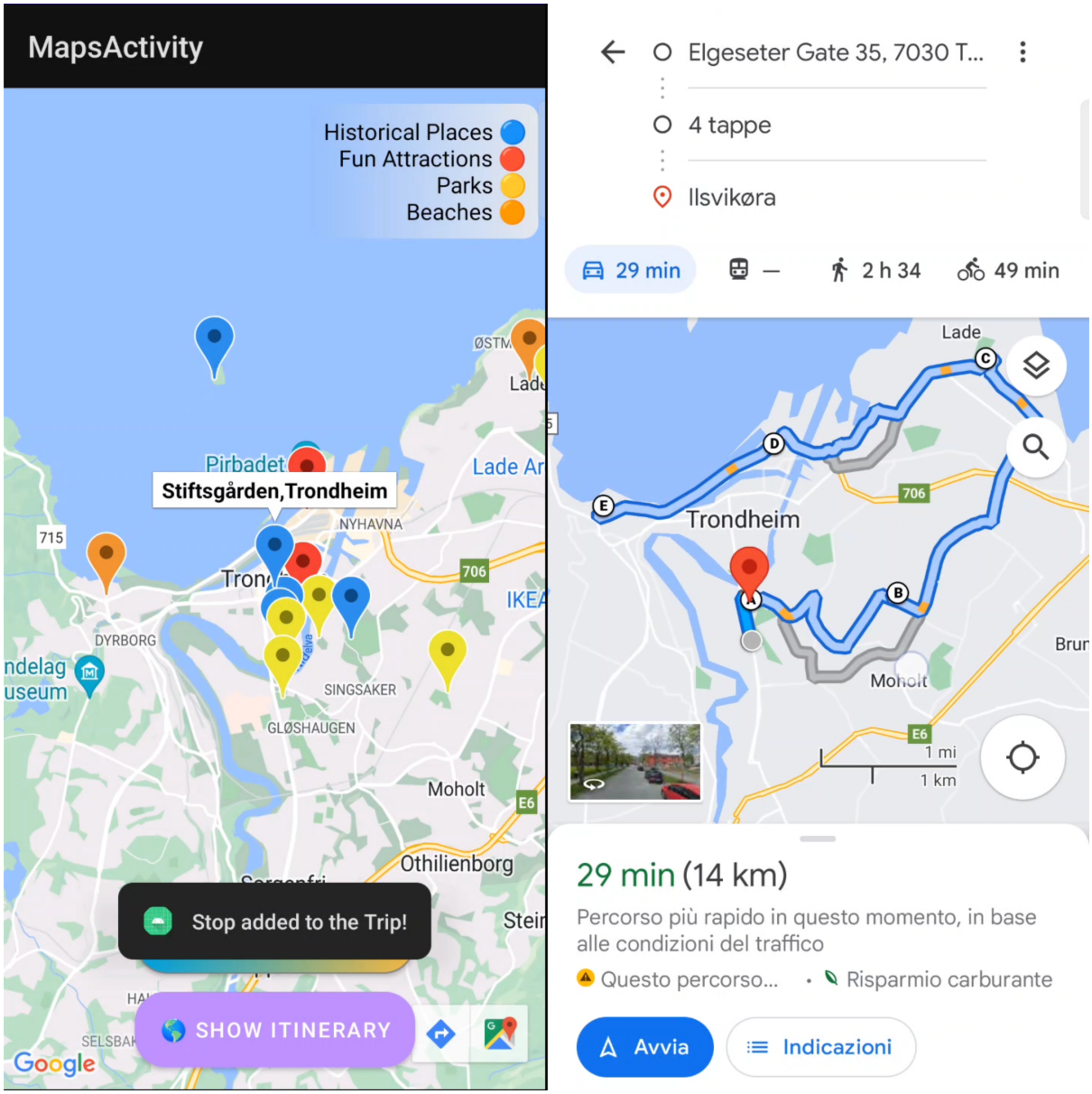


6. The application can also display a map showing the locations of the different places.
7. The user can now:

   7A. Select one of the places and either see it on google maps or receive directions for the single place.



7B. Add one or more places from the map and can be redirected to a Google Map itinerary.



Technical Challenges

Our project was faced with several technical challenges, in particular:

1. **GPS Sensor for Geo-localization:** One of the main challenges we faced was the need to accurately locate our users using a GPS sensor that can reliably determine a user's position and return `(x, y)` position.
2. **ChatGPT API Integration for gathering results:** Our project relies heavily on the use of ChatGPT API for gathering results. We needed to ensure that we have a seamless integration with the API, tuning the query to work properly and optimizing the costs (longer queries cost more money).
3. **Google Maps for map visualization:** To provide users with an intuitive and interactive view of their location, we needed to integrate Google Maps into our app.

   This required us to use an API that can display maps and other location-related data.

The most difficult part was highlighting our results points in the map, doing it in an efficient way.

4. **Itinerary**: The last step was implementing a quick and efficient way to redirect the user to its own built itinerary.

GPS Sensor - Geo-Localization

The location of the user is gathered in the main activity, in order to avoid making this extra step when querying GPT, which is already a computationally demanding activity.

In order to get the location permissions must be granted; then, when the location is ready, the corresponding field will be filled and everything will be used in the query to GPT.

In order to increase readability a class `Point` is built to contain `x (latitude)` and `y (longitude)` coordinates.

User location, together with other properties, are then stored in `Place` objects, corresponding to the found top places to visit.

Google Maps instead uses its own `LatLng` class, which will be stored as information as well.

In this case, we have some redundancy but is needed to handle the interaction with the Google API.

Distance Calculation:

In more than one case, the distance between points on the earth's surface must be computed.

We rely on a well-known formula, included in a function in the `Point` class.

ChatGPT API Integration - Gathering Results

This was one of the most complex steps, which included:

- understanding the library (***GptChatApiService***),
- implementing async tasks in order to not overload the main UI thread,
- tuning the query,
- retrying the query if it failed,
- parsing results and filling data structures.

The library provides a way to interact with ChatGPT API.

It has a wide range of possibilities, in particular, different models and different capabilities; we used the *gpt 3.5 turbo* model and exploited the chat completion functionality.

Our queries are built at runtime taking into input the information provided by the user:

```
Tell me the NUM_OF_RESULTS best CATEGORY near these coordinates (X, Y)
in the range of RANGE kilometres
....
```

The number of results was initially 10, but the API seemed to take too much responding and the query had a **low success rate**; taking only half results instead made the query response faster and easier to be produced correctly by the model.

Once the query is ready, it is sent to GPT.

We have one query for every category selected by the user and each query is processed by a different async task.

At this point, when the response is available or when the query generates an error, different functions are triggered to handle both cases.

In particular, we have three possible outcomes:

- ***success***: the response is correct and once parsed it provides at least one place to visit.
- ***unsuccess***, divided in:
  - the query to GPT fails and returns an error.
  - the response is correct but once parsed it doesn't provide a place to visit.

  In both cases, the query is retried for a maximum of 2 times, and then an empty result is returned.

Parsing the response consists of filling in the class `TopPlaces`, which contains information about the top places to visit related to a certain category.

In the case of ***success***, this class will contain the response and a list of `Place` (s), which are basically the result of the parsing.

For example, consider the response:

```
1. Paraggi Beach - Genova, Italy - 44.319652, 9.195813 - Small, exclusive cove with turquoise waters and rocky cliffs.
....
```

The first line will be converted into an object `Place` containing:

- name: `Paraggi Beach`,
- city: `Genova, Italy`,
- position: `44.319652, 9.195813`,
- short description: `Small, exclusive cove with turquoise waters and rocky cliffs`.

Otherwise, if the query ends with an ***unsuccess***, the field `retries` is incremented until a maximum of three.

Google Maps - Map Visualization

To implement the Map Visualization we simply follow the **Maps SDK for Android Quickstart:**

- https://developers.google.com/maps/documentation/android-sdk/start?hl=en

After that, the Maps were visualized in the correct way we start to add the markers on the map.

For Each Marker's Category, we choose a different Color:

- Historical Places
- Parks
- Beaches
- Fun Attractions

On the marker’s click, it shows a TextView representing the title of the marker and the *Add Stop Button* which will add the stop to the itinerary.

We encountered some issues when trying to place markers on the map because **the coordinates provided by ChatGPT were not very accurate**. As a result, we searched for an alternative solution to find more precise coordinates and discovered the ***Google Geocoder library***. This library allows us to retrieve the coordinates of a location based on its name. The following solution proves to be extremely precise for places with specific names (e.g., "Acquario di Genova"), while for places with less specific names, the Geocoder API call might not return any results or the result couldn’t be accurate.

Consequently, **we opted for a hybrid implementation** where, whenever possible, we use Geocoder to obtain precise coordinates. However, if Geocoder doesn't return any results, we fall back to the coordinates provided by ChatGPT.

The first implementation was done using directly the UI Thread, but we immediately noticed that when the markers were too much the application crashed.

As a consequence we used an AsyncTask for each category in this way we resolve the problem of the UI Thread and the application didn’t crash anymore.

Google Maps - Itinerary

To create the itinerary, we had two possible options:

1. Implement it in the MapsActivity.
2. Utilize the Google Maps Application.

Regarding the first solution, we didn't gather enough information on how to implement it, and it turned out to be too complex.

On the other hand, the second solution simplified the implementation of this functionality significantly. Once we discovered how the Google Maps link was constructed:

- https://www.google.com/maps/dir**/x,y/stop+1,+City,+State/stop+2,+City,+State/**

The next step was very easy. We simply raised an intent with the following link, and Google Maps did all the rest of the work, creating an itinerary.

To choose what Stop we want to add to the itinerary, we implement a button that appears every time we click on the marker, in this way we can add the stop to an array that will be parsed to create the link above.